Towards a WebAssembly Standalone Runtime on GraalVM

Salim S. Salim
University of Manchester
Manchester, UK
salim.salim@manchester.ac.uk

Andy Nisbet
University of Manchester
Manchester, UK
andy.nisbet@manchester.ac.uk

Mikel Luján
University of Manchester
Manchester, UK
mikel.lujan@manchester.ac.uk

Abstract

WebAssembly is a binary format compilation target for languages such as C/C++, Rust and Go. It enables execution within Web browsers and as standalone programs. Compiled modules may interoperate with other languages such as JavaScript, and use external calls (imports) to interact with a host environment. Such interoperability dependencies influence the overall WebAssembly module performance and can limit Web/standalone execution capabilities.

The implementation of a WebAssembly runtime, called TruffleWasm is described that provides a single environment for execution of both, standalone modules, and, interoperation with multiple GraalVM hosted languages such as JavaScript (GraalJS) via Truffle's interoperability framework. The *Graal* compiler is used to speculatively and aggressively apply profiling driven optimisations to perform *Just-in-Time (JIT)* code generation.

CCS Concepts • **Software and its engineering → Interpreters**.

Keywords WebAssembly, Wasm, WASI, Truffle framework, Graal, TruffleWasm

ACM Reference Format:
Salim S. Salim, Andy Nisbet, and Mikel Luján. 2019. Towards a WebAssembly Standalone Runtime on GraalVM. In *Proceedings of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '19), October 20–25, 2019, Athens, Greece.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3359061.3362780

1 Introduction

WebAssembly is a binary format designed to interoperate with JavaScript and work inside JavaScript engines [2, 5]. Programs written in languages such as C/C++, Rust and Go can be compiled to target WebAssembly and interoperate with JavaScript. Each program is compiled into a single unit called a `module`. A module is composed of global definitions such as `table`, `memory`, `imports`, `exports`, `types` and `functions`. `imports` define a list of global elements that should be provided by the runtime (either implemented by the runtime or loaded from another module). `exports` define a list of internal elements that can be accessed outside the module. The `memory` is a linear array of uninterpreted bytes accessed by `load`/`store` instructions and other API functions [5].

While its primary motivation was the Web, WebAssembly has been ported to run outside the Web as a standalone compilation target or embedded with other language frameworks such as Rust and Go. The WebAssembly C/C++ compilation pipeline uses LLVM [4] infrastructure to produce WebAssembly binaries without targeting a specific API. Nevertheless, there have been efforts to standardise how WebAssembly modules interface with host environments. *WebAssembly System Interface (WASI)* is a standardisation for a modular system interface for WebAssembly overseen by a special W3C subgroup[1]. It defines a set of *POSIX/CloudABI* like APIs that a runtime can implement to run WebAssembly modules as standalone applications. Another ongoing proposal is the *WebAssembly Interface Types*[2], that allow WebAssembly modules to interoperate with abstract types from other languages (or modules) using a single interface.

Truffle can be used to build *Abstract Syntax Tree (AST)* interpreters on the *Java Virtual Machine (JVM)*. Truffle interpreters can benefit from the wide range of optimisations supported by the Graal compiler that exploits profiling information from interpreted execution to aggressively specialise and optimise code-generation from an AST [6]. The Truffle framework also provide infrastructure for efficient interoperability between languages hosted on GraalVM, and a Native Function Interface (*TruffleNFI*) to allow interpreters to efficiently access native functions [1].

We present the first implementation of a WebAssembly runtime on GraalVM using the Truffle framework called TruffleWasm. TruffleWasm executes WebAssembly modules and exploits partial evaluation and JIT compilation using the Graal compiler. It is designed to support WebAssembly modules compiled to target different environments such as WASI, JavaScript interoperability or standalone programs.

SPLASH Companion '19, October 20–25, 2019, Athens, Greece

ACM ISBN 978-1-4503-6992-3/19/10.
https://doi.org/10.1145/3359061.3362780

[1] https://wasi.dev/
[2] https://github.com/WebAssembly/interface-types/

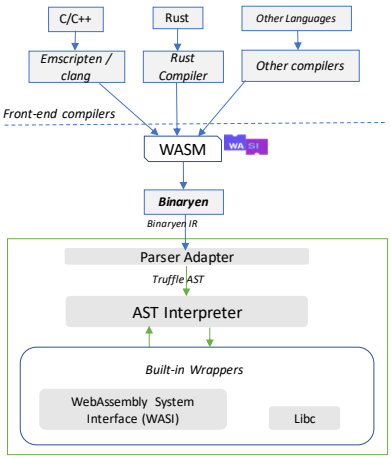

Figure 1. TruffleWasm: key runtime components. Parsing uses Binaryen[3] to produce an *Abstract Syntax Tree* (AST) IR from a WebAssembly module.

2 Design and Implementation

TruffleWasm implements the core specification (from the *Minimal Viable Product (MVP, i.e v1.0)*, WASI API and a prototype of a subset of standalone *libc* functions. The core WebAssembly instructions are implemented as Truffle Nodes. The interpreter provides runtime specialisation, profiling and compiler annotations to be used by the Graal compiler for partial evaluation and JIT code generation. Figure 1 illustrates the design of our runtime that can execute modules compiled to target WASI API, or to interoperate with other languages such as JavaScript.

Linear memory Is implemented using the Java `Unsafe` API and is bounds checked to ensure modules only access memory within their assigned boundaries. The linear memory object exposes specific methods that are used by `load` and `store` instructions and other built-in functions. Linear memory is used as a way of sharing information between functions, maintaining struct/object states and inter-operating with other modules. As such, WebAssembly modules contain more `loads` and `stores` than their native counterpart [3], and it is therefore important to be optimised for efficient execution of WebAssembly programs.

WASI APIs Each WASI function is designed as a Truffle `Node` built-in function that uses *TruffleNFI* to call a sand-boxed *POSIX* function which is checked according to the WASI specification.

libc API Prototype Compiler tools such as `Emscripten` and `WasmExplorer` allow compiling to WebAssembly standalone modules without targeting any specific API[4]. Standalone modules are compiled through LLVM and external functions are left in the module as `imports`. For a runtime to execute these modules, it provides built-in support for any imported functions. We implemented the import functions generated when compiling small applications (such as Shootout benchmarks), and are able to run them as standalone programs.

Interoperability with other Languages When using Truffle languages, it is possible to load a WebAssembly module and call its exported functions. This is done through Truffle interoperability and it allows any Truffle language to interoperate with WebAssembly. For JavaScript's `WebAssembly.*` API, we have prototype implementation for basic loading and initialisation of WebAssembly modules from GraalJS.

3 Future Work

Adding features beyond MVP New proposed features which are currently being standardised (some are already in browsers) are our next target to support in the future.

Complete API to support large code-bases and programs TruffleWasm can currently run programs from *PolybenchC* and *Shootout* benchmark suites. Current work aims to provide full WASI API support for a wider range of applications.

Interoperability with JavaScript and other Languages We are in the process of providing a complete JavaScript `WebAssembly.*` API through GraalJS and aim to study the performance of interoperability using Truffle Interop.

Acknowledgments

Mikel Luján is funded by an Arm/RAeng Chair Award and Royal Society Wolfson Fellowship and Andy Nisbet is funded by the EU H2020 ACTiCLOUD 732366 project.

References

[1] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance Cross-language Interoperability in a Multi-language Runtime. *SIGPLAN Not.* 51, 2 (Oct. 2015), 78–90. https://doi.org/10.1145/2936313.2816714

[2] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 185–200. https://doi.org/10.1145/3062341.3062363

[3] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: Analyzing the Performance of WebAssembly vs. native code. In *Annual Technical Conference ({USENIX}{ATC} 19)*. 107–120.

[4] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[5] W3C. 2018. WebAssembly Core Specification. Retrieved 2018-02-15 from https://www.w3.org/TR/wasm-core-1/

[6] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 662–676. https://doi.org/10.1145/3062341.3062381

[3] https://github.com/WebAssembly/binaryen
[4] https://github.com/kripken/emscripten/wiki/WebAssembly-Standalone